An exact algorithm for the constraint satisfaction problem : application to dependance computing in automatic parallelization

H. Bennaceur, G. Plateau, F. Thomasset

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1246

Programme 2
Structures Nouvelles d'Ordinateurs

AN EXACT ALGORITHM FOR THE CONSTRAINT SATISFACTION PROBLEM : APPLICATION TO DEPENDANCE COMPUTING IN AUTOMATIC PARALLELIZATION

Hachemi BENNACEUR
Gérard PLATEAU
François THOMASSET

Juillet 1990

RR-1246

An exact algorithm for the constraint satisfaction problem: Application to dependance computing in Automatic parallelization

Un algorithme exact de résolution du problème de satisfaction de contraintes: application au calcul de dépendances en parallélisation automatique

Hachemi BENNACEUR, Gérard PLATEAU *
François THOMASSET †

Juin 1990

*Université Paris-Nord, Laboratoire d'Informatique de Paris Nord CSP - avenue J.B. Clément 94430 VILLETANEUSE
†INRIA - BP105 - F-78153 LE CHESNAY

An exact algorithm for the constraint satisfaction problem: Application to dependance computing in automatic parallelization

Hachemi BENNACEUR, Gérard PLATEAU, François THOMASSET

Abstract:

the contraint satisfaction problem -denoted by CSP- consists in proving the emptiness of a domain defined by a set of constraints or the existence of a solution. Numerous applications arise in the computer science field (artificial intelligence, vectorization, verification of programs, ...).
In the case of the study of dependence computing in automatic parallelization, classical methods in literature may break down for some instances of CSP, even with small sizes. By constrast the new method -denoted by FAS^{3}T (Fast Algorithm for the Small Size constraints Satisfaction problem Type)- we propose allows an efficient solution of the CSP concrete instances generated by the VATIL vectorizer. Comparative computational results are reported.

Résumé:

le problème de satisfaction de contraintes -noté CSP- consiste à prouver la vacuité du domaine défini par un système de contraintes ou l'existence d'une solution. Ce problème se rencontre dans de nombreuses applications informatiques (intelligence artificielle, vectorisation, vérification de programmes,...).
Dans le cadre du calcul de dépendances pour la parallélisation automatique, il existe des instances du CSP même de tailles modestes pour lesquelles les méthodes classiques de la littérature aboutissent à une indécidabilité.
A l'inverse, la nouvelle méthode proposée -notée FAS^{3}T (Fast Algorithm for the Small Size constraints Satisfaction problem Type)- permet la résolution efficace du CSP pour les problèmes concrets générés par le vectoriseur VATIL. Des expériences numériques comparatives sont détaillées.

Key Words : Satisfaction of constraints - automatic vectorization - integer programming - data flow dependency.

1 Introduction

The core element of an **automatical vectorizer** is the study of data flow dependency inside a program. It consists in analysing the data used or produced by a statement with the aim of accepting or rejecting the vectorization.

This problem can be modelizing as a system of diophantine constraints for which the existence of a solution or the emptiness of the associated domain has to be proved; only in this second case, the concerned part of code can be vectorized.

As this **constraint satisfaction problem** -denoted by **CSP**- is NP-complete, most of the authors' studies are devoted to approximate solving technics (approximate decidability): in fact continuous linear constraint systems are solved after relaxing the integrality conditions on the variable of the concrete problem [Bledsoe 74,75, Shostack 77,81],

Although the use of classical linear programming algorithms (simplex [Dantzig 63] or projective [Karmarkar 84] methods) may lead to exact solutions for some instances of the CSP (emptiness of the continuous domain, or unimodularity property satisfied, or detection of an integer solution during the resolution), it remains that when the CSP instance is satisfiable on $\mathbb{R}$ it is impossible to conclude the satisfiability or the nonsatisfiability on $\mathbf{Z}$; in this general case, the part of associated code cannot be vectorized.

This paper is organized as follows:

Section 2 gives the principles of dependence computing as implemented in the current **VATIL vectorizer** [Thomasset 88]. A preprocessing scheme including the GCD and Banerjee-Wolfe tests, and linearizations is used before to perform the simplex method.

Section 3 deals with the description of an **exact algorithm** -denoted by **FAS^{3}T** (Fast Algorithm for the Small Size constraint Satisfaction Test problems)- for the **CSP**. Thus the aim of this general method [Bennaceur 89, Bennaceur, Plateau 89a and 89] is the proof of the emptiness of the domain defined by any system of diophantine constraints, or the existence of a solution.

Section 4 details the computational results on a SUN 3/160 computer with a Fortran 77 implementation of FAS^{3}T performed with a lot of concrete instances generated by the VATIL vectorizer.

2 Dependence computing in automatic parallelization

We give the principles of dependence computing as implemented in the current VATIL vectorizer [Thomasset 87]. These are exposed for the sake of simplicity in the case of simply nested loops, but they may be easily extended to multiply nested loops.

2.1 the problem

Consider a FORTRAN loop:

```
      DO 1 I = 1, N
A::       X(a1 * I + a0) = ...
B::       ...          = X(b1 * I + b0) ...
1     CONTINUE
```

We have to answer the following question:

> Does there exist values of the index i j,
> subject to loop bounds
> such that A(i) (at iteration i) and B(j) (at iteration j)
> access the same memory cell.

This is represented by a set of equations of the following form:

$$a_1 * i + a_0 = b_1 * j + b_0$$
$$1 \leq i \leq N$$
$$1 \leq j \leq N$$
$$i \ op \ j$$

in which op is either <, > or = depending whether we look for true-, anti-, or loop independent dependences:

Let us give a few examples:

Example 1:

```
    DO 1 I = 1, 10
      X(2*I) = 1
      Y(I)   = X(2*I+1)
1   CONTINUE
```

It is easily checked by inspection of index expressions that there is no dependence here, since the first instruction accesses the even positions of array X, and the other one, the odd positions: this is an example of application of classical ``GCD test".

Example 2:

```
    DO 1 I = 1, 10
      X(3*I+10) = 1
      Y(I)    = X(I)
1   CONTINUE
```

Here the absence of dependence will result from consideration of the values of loop bounds: there exists no solution to the system:

$$3 * i + 10 = j$$
$$1 \leq i \leq 10$$
$$1 \leq j \leq 10$$

Example 3:

```
    DO 1 I = 1, 10
      X(M*I+N) = 1
      Y(I)    = X(I)
1   CONTINUE
```

Now we have no possibility to say anything about the existence of dependences unless somcthing is known about the variables M and N: indeed if it were known that M and N take on positive values at execution then we could assert the absence of dependence; this kind of information may come either from automatic data flow analysis or from assertions given by the programmer: in VATIL we have implemented the possibility to assert bounds and linear relations about the variables; these rclations will be added to system; it is the subject of this paper to describe the exploitation of such information.

It is worthwhile to notice that the system in the analysis of example 3 is non linear:

$$M^* i + N = j$$
$$1 \leq i \leq 10$$
$$1 \leq j \leq 10$$
$$i \ op \ j$$

Since the general integer programming problem is known to be undecidable, the brute solution of such a system is unfeasible, and approximations will have to be made in order to come to a linear system, allowing an approach to an approximate solution: the well known Banerjee test is one such test (see further below); of course such approximations may lead to imprecise results and find spurious dependences. The vectorizer will make conservative assumptions and declare dependence whenever the absence of dependence cannot be proved.

Our strategy is exposed below as a succession of tests:

first try the GCD test or symbolized GCD test;
then form the inequations for the Banerjee test using informations from data flow analysis or programmer assertions;
if these inequations cannot be numerically evaluated, then:

linearize the system;
invoke a procedure such as the simplex method in order to try to prove the absence of feasible solution;
if the procedure finds that solutions may exist, decide the existence of a dependence.

Note that we make approximations at several levels:

using the Banerjee test, since we relax the constraint on integrity of solutions;
during the linearization process;
possibly on invoking an algorithm such as the simplex since the standard simplex method works on system with real variables.

2.2 The GCD test

This test looks at the index expression without considering the loop bounds; the equation to be solved is of the form:

$$a_1 * i - b_1 * j = (b_0 - a_0)$$

where the a_1, a_0, b_1, b_0, i, j, must take on integer values; then it is straightforward to check that a necessary condition for existence of a solution is that:

the GCD of a_1 and b_1 divides $(b_0 - a_0)$.

This test is easily conducted when the coefficients a_1, b_1 and the difference (b_0-a_0) are all integer constants; otherwise we can easily perform symbolic manipulations to determine integer factors of expressions:

```
find (exp)
    --- returns a number in any case
    --- exp is an integer multiple of find(exp)
        CASE exp OF:
            number : return exp
            e1 * e2 : return find(e1) * find (e2)
            e1 + e2 : return GCD (find(e1) , find (e2))
            e1 - e2 : return GCD (find(e1) , find (e2))
            other cases : return 1
        END CASE
END find
```

Example:

find (2 * x + 4* (y + z)) = 2

Then the symbolized GCD test becomes:

if $b_0 - a_0$ is a number,
the GCD of find(a_1) and find(b_1) divides ($b_0 - a_0$).

No conclusion can be drawn when ($b_0 - a_0$) is not a number.

2.3 Banerjee Test

We relax the constraint on integrity of solutions in order to reduce the complexity of the problem [Banerjee 76]:

let p and q [1] be the bounds of the current loop:

```
      DO 1 I = p,q
      ...
1     CONTINUE
```

given any integer expressions a and b we set:

$f_{a,b}(x,y) \equiv a * x - b * y$

the given problem is to search a solution x,y of:

$f_{a1,b1}(x,y) = b_0 - a_0$

in regions of the iteration space:

either the triangle (op is <):

$p \leq x < y \leq q$

or the triangle (op is >):

$p \leq y < x \leq q$

or the line (op is $=$):

$p \leq x = y \leq q$

---

[1] which may be expressions; classical presentations of this test require p to be 1

compute lower and upper bounds of $f_{a1,b1}(x,y)$ in the relevant region according to formulae below;

- If $b_0 - a_0$ is between the computed bounds for $f_{a1,b1}(x,y)$ this implies the existence of a solution in $\mathbb{R}^2$; then assume a dependence; otherwise there is no dependence.

We now have to compute the bounds of $f_{a1b1}(x,y)$; these depend on the signs of the coefficients a_1 and b_1 and of the difference $a_1 - b_1$.

Bounds of $f_{a,b}$ in the triangle of plane x,y defined by:

$p \leq x < y \leq q$

	LOWER BOUND	UPPER BOUND
$a \geq 0, b \geq 0$	$f_{a,b}(p,q)$	
$a \geq b$		$f_{a,b}(q-1,q)$
$a \leq b$		$f_{a,b}(p, p+1)$
$a \geq 0, b \leq 0$	$f_{a,b}(p,p+1)$	$f_{a,b}(q-1, q)$
$a \leq 0, b \leq 0$		$f_{a,b}(p,q)$
$a \geq b$	$f_{a,b}(p,p+1)$	
$a \leq b$	$f_{a,b}(q-1,q)$	

Bounds of $f_{a,b}$ in the triangle of plane x,y defined by:

$p \leq y < x \leq q$

	LOWER BOUND	UPPER BOUND
$a \geq 0, b \geq 0$		$f_{b,a}(p,q)$
$a \geq b$	$f_{b,a}(q-1,q)$	
$a \leq b$	$f_{b,a}(p,p+1)$	
$a \geq 0, b \leq 0$	$f_{b,a}(q-1,q)$	$f_{b,a}(p,p+1)$
$a \leq 0, b \leq 0$	$f_{b,a}(p,q)$	
$a \geq b$		$f_{b,a}(p,p+1)$
$a \leq b$		$f_{b,a}(q-1,q)$

Bounds of $f_{a,b}$ along the line defines by:

$p \leq x = y \leq q$

	LOWER BOUND	UPPER BOUND
$a \geq b$	$(a-b) * p$	$(a-b) * q$
$a \leq b$	$(a-b) * q$	$(a-b) * p$

2.4. Finding the signs of coefficients

It has been noted that the bounds of a linear function depends on the sign of its coefficients; when these happen to be non numbers the pure Banerjee test gets stuck; in order to circumvent this, we use the following heuristics: we perform a preliminary pass in the set of data flow informations and pick up any relation of the form:

V op N

with V a variable and N a number.

This information is used to compute bounds on V and expressions using V.

Of course it will happen that some variables are involved whose bounds cannot be determined; then the test can be only partially conducted; for instance if we know that a_1 and b_1 are positive, but ignore the sign of $b_1 - a_1$ we can find the lower bound of $f_{a,b}$ on the triangle $x<y$ but not the upper bound.

2.5. Dependence Computing: linearization

Clearly the inequations from the Banerjee test can be non linear; therefore we enter any inequation coming either from the Banerjee test or from data flow information into a *linearization* procedure, this creates new variables to represent the non linear terms and substitutes in the expressions; therefore the procedure receives an argument representing the expression to be linearized, and maintains a global variable which is the current set of new symbols.

```
lin (e)
    CASE e :
        symbol or number : return e
        e1 +e2 : return lin(e1) +lin(e2)
        e1 - e2 : return lin(e1) - lin(e2)
        - e1 : return - lin(e1)
        e1 * e2 :
            let n1 = lin (e1), n2 = lin(e2) in
                if n1 is a number or n2 is a number
                    then return n1 * n2
                    else return newsymb (e)
        other cases: return newsymb (e)
    END CASE
end lin
```

3 Method FAS^{3}T for the exact solution of the CSP

This part deals with the description of a new exact algorithm - denoted by FAS^{3}T (Fast Algorithm for the Small Size constraint Satisfaction Test problems) - for the constraint satisfaction problem. The principle of the method (section 3.1) and the basic theoretical results (section 3.2) precede the description of the algorithm (section 3.3). A numerical example illustrates the method in section 3.4. The numerical experiments devoted to this type of system:

$$(S) \qquad Ax \leq b \; ; x \in Z^n$$
$$\text{with } A \in Z^{mxn} \text{ and } b \in Z^m$$

(automatical program vectorization CSPs' model) are detailed in section 3.5.

3.1 Principle of the method

FAS^{3}T is a new method (for more details see [Bennaceur 89, Bennaceur, Plateau 89a and 89b]) for the exact solving of the constraint satisfaction problem (CSP). Its aim is the proof of the emptiness of a domain defined by a system of diophantine constraints , or the existence of a solution.
FAS^{3}T is a general method which can be used for any type of systems; but due to the NP-Completeness of the CSP, it has to be performed for adequate diophantine systems (particular structures, sizes not too large,...).

Given a diophantine system

(S) $Ax \leq b$; $x \in D$
where A is a mxn matrix and D is a discrete set (Z^n, N^m, $\{0,1\}^n$),

the scheme of FAS^{3}T consists in generating a finite sequence of integer points

$x^0, x^1, x^2, \ldots, x^k$ where $k \leq m$

until

either x^k satisfies the system of constraints S
or the associated domain $F(S)=\{x \mid Ax \leq b;\ x \in D\}$ is proved to be empty.

Each integer point x^h ($h \in \{1,...,k\}$) is an optimal and feasible solution (or the current best feasible solution for x^k) of an integer programming problem with this form :

$$(P^h) \quad \begin{array}{ll} \min & g^h(x) \\ \text{s.t.} & A_i x \leq b_i \qquad \forall i \in I^h \subset \{1,...,m\} \\ & x \in D \end{array}$$

Given a starting point x^0, the problems (P^h) h=1,...,k-1 satisfy the following properties :

(i) I^h is the subset of constraints of S satisfied by x^{h-1}

(ii) the function g^h depends on $\bar{I}^h = \{1,...,m\} \setminus I^h$, the subset of constraints of S non satisfied by x^{h-1}. It is such that x^h satisfies at least one new constraint of the system S (i.e. $I^h \subset I^{h+1}$ with $|I^{h+1}| \geq |I^h| + 1$).
Thus, method FAS^{3}T solves a sequence of k ($k \leq m$) n-integer variable optimization problems whose number of constraints increases from a problem to the following one, but never exeeds m-1.

Note: when the domain of (P^h) is unbounded, if an extreme direction is found during optimization, the adding of a constraint of the type $cx \geq -M$ (with the positive integer number M sufficiently large) allows to ensure the finiteness of $v(P^h)$.

3.2 Basic theoretical results

Given h in $\{1,...,k\}$, by taking into account the current choice of the function g^h:

$$g^h(x) = \sum_{i \in \bar{I}^h} A_i x; \quad x \in D,$$

the following results give stopping criteria for the method FAS^{3}T: (proofs in [Bennaceur 89, Bennaceur, Plateau 89a])

Theorem 1:

If one of the following conditions holds

(i) $v(P^h) > \sum_{i \in \bar{I}^h} b_i$

(ii) $v(P^h) = \sum_{i \in \bar{I}^h} b_i$, and

$\forall\ x^*$ optimal solution of (P^h) : $\exists\ i \in \bar{I}^h$ such that $A_i\ x^* \neq b_i$

then F(S) is empty.

Theorem 2:

If the following condition holds :

$$A_i\ x^h \leq b_i \qquad \forall\ i \in \bar{I}^h$$

then $x^h \in$ F(S)

If the conditions of theorems 1 and 2 do not hold, then the following results are used:

Corollary 1:

(i) $v(P^h) \leq \sum_{i \in \bar{I}^h} b_i$

(ii) $\exists\ i \in \bar{I}^h : A_i\ x^h > b_i$ and $\exists\ j \in \bar{I}^h : A_j\ x^h < b_j$

Let us denote

$$J^h = \{\ i \in \bar{I}^h \mid A_i\ x^h \leq b_i\ \}$$

by corollary 1. (ii), $1 \leq J^h \leq I^h$, and

Theorem 3:

Given $h \in \{1,...,k-2\}$, and x^h (resp. x^{h+1}) the feasible optimal solution of (P^h) (resp. (P^{h+1})), then

$$F(S) \neq \emptyset \Rightarrow \sum_{i \in J^h} A_i x^{h+1} > \sum_{i \in J^h} A_i x^h$$

Corollary 2:

If

$$\forall i \in J^h \quad A_i x^h = b_i$$

then $F(S)=\emptyset$

3.3 Algorithm FAS^{3}T:

The algorithm FAS^{3}T for the exact solving of the constraint satisfaction problem is based on the theoretical results described in section 3.2.

empty ← false; {the boolean "empty" is true when the emptiness of F(S) is proved}
end ← false; {the boolean "end" is true when an element of S is found}

choose x^0 in D;
if $x^0 \in F(S)$ **then** end ← true
else
I ← $\{i \in \{1,...,n\} \mid A_i x \leq b_i\}$; $\bar{I}$ ← $\{1,...,m\} \setminus I$;
solve

(P) min $\sum_{i \in \bar{I}} A_i x$ s.t. $x \in D \cap D'$

where D' is such that $v(P) \in \mathbf{Z}$
and $F(P) \cap F(S) \neq \emptyset \vee F(S)=\emptyset$; {note of section 3.1}

x^* ← optimal solution of (P);
while ¬ empty **and** ¬ end **do**

```
if v(P) > Σ_{i∈Ī} b_i then empty ← true                    {theorem 1 (i)}
else J ← { i ∈ Ī | A_i x* ≤ b_i } ;
     if J = Ī then end ← true                              {theorem 2 }
     else if x* is unique and v(P)= Σ_{i∈Ī} b_i
          then empty ← true                                {theorem 1 (ii)}
          else I ← I ∪ J ; Ī ← I \ J ;
               solve
               (P) min Σ_{i∈Ī} A_i y s.c. A_i y ≤ b_i  i∈I, y∈D∩D'
               if an element of F(S) is found by solving (P)
               then end← vrai
               else
                    y* ← optimal solution of (P);
                    if Σ_{i∈J} A_i y* ≤ Σ_{i∈J} A_i x* then
                    empty ← true     { theorem 3 }
                    else x* ← y*
                    endif
               endif
          endif
     endif
endwhile
endif
```

Proposition:

The number of iterations of algorithm FAS^{3}T is bounded by m

Proof:

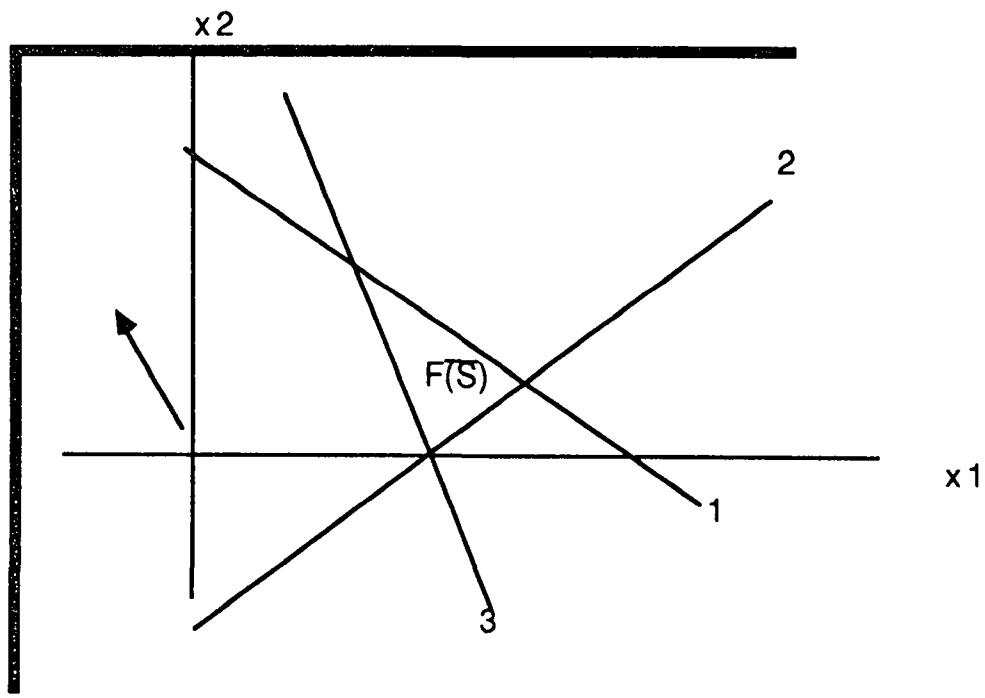
Direct consequence of corollary 1 which proves that at each iteration $|J| \geq 1$ and thus $|\bar{I}|$ decreases by at least one unity.

In the worst case the algorithm stops with the solving of a problem whose objective function is reduced to one constraint of S subject to the m-1 other constraints of S.

3.4 Numerical example:

$$
(S)\quad
\begin{array}{lll}
2x_1+3x_2 & \leq 6 & (1)\\
2x_1-3x_2 & \leq 3 & (2)\\
-3x_1-x_2 & \leq -5 & (3)\\
x1,\ x2 \in \mathbb{Z} & &
\end{array}
$$

(let $F(\bar{S})$ denote $\{x \in \mathbb{R}^2 \mid x$ satisfies (1), (2) and (3) $\}$



initialization

the starting point x^0 may be choosen by solving

$$
(P^0)\quad
\begin{array}{ll}
\min & x_1-x_2\\
\text{s.t.} & x \in D'
\end{array}
$$

$$x_1, x_2 \in \mathbb{Z}$$

where $D' = \{x \in R^2 \mid x_1 \geq -10;\ x_2 \leq 10\}$

whose value and solution are

$v(P^0) = -20$ and $x_1^* = -10$, $x_2^* = 10$

thus

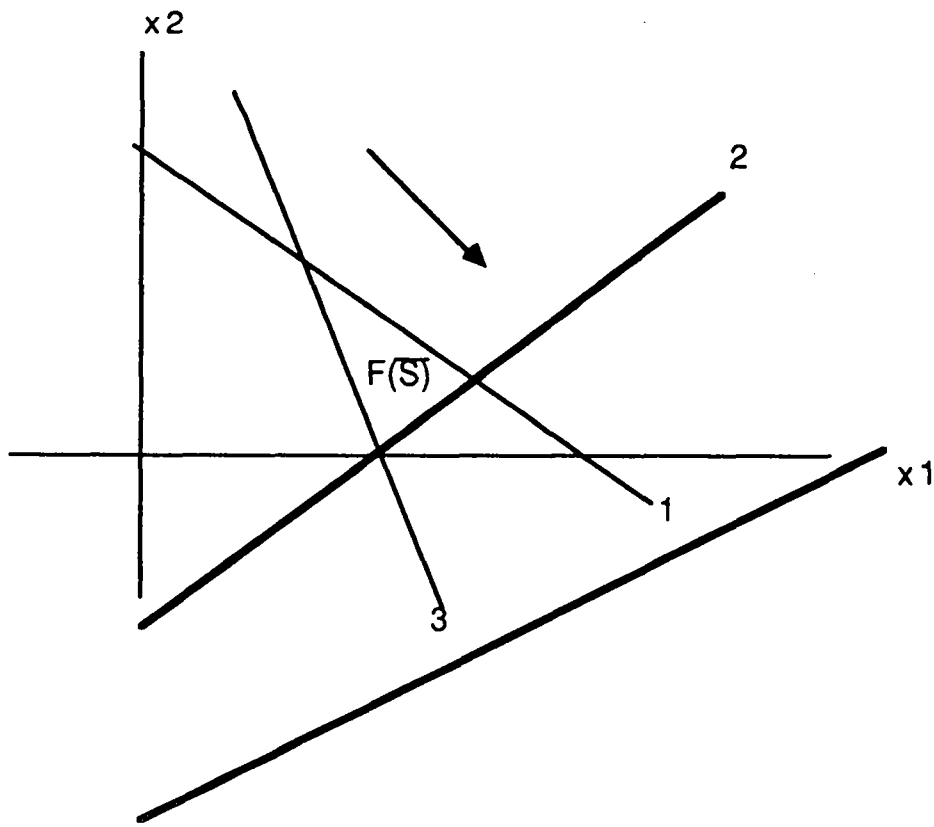
$I = \{2\}$ and $\bar{I} = \{1,3\}$

note: as the right hand-side of constraints (1) and (2) are non negative, the initial point x0 might have been the origin. But we have taken the above choice for the sake of explanation in order to show several iterations.

Iteration 1

$$(P^1) \quad \begin{array}{ll} \min & -x_1 + 2x_2 \\ \text{s.t.} & 2x_1 - 3x_2 \leq 3 \\ & x \in D' \\ & x_1, x_2 \in \mathbb{Z} \end{array}$$

where $D' = \{x \in R^2 \mid -x_1 + 2x_2 \geq -10\}$



the value and solution of (P^1) are

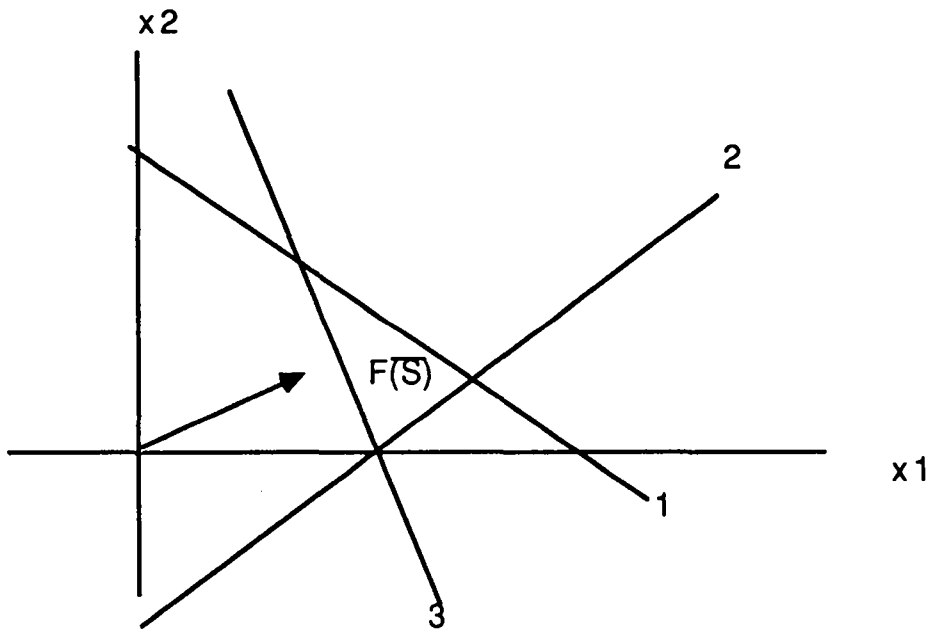
$v(P^1)=-13$ and $x_1^*=-33$, $x_2^*=-23$

thus

$I=\{1,2\}$ and $\bar{I}=\{3\}$

iteration 2

$$(P^2)\quad \begin{array}{ll} \min & -3x_1-x_2 \\ \text{s.t.} & 2x_1+3x_2 \leq 6 \\ & 2x_1-3x_2 \leq 3 \\ & x_1,x_2 \in Z \end{array}$$



the value and solution of (P^2) are

$v(P^2)=-4$ and $x_1^*=1$, $x_2^*=1$

the first condition of theorem 1 holds, thus F(S) is empty.

4 Numerical experiments

The Fortran77 code of algorithm FAS^{3}T has been implemented on a SUN 3/160 computer with a lot of concrete instances of this type

$Ax \leq b;\ x \in Z^n$
with $A \in Z^{mxn}$ and $b \in Z^m$

generated by the vectorizer VATIL [Thomasset 87].

4.5.1 Resolution of the current problem

The current problem generated at each iteration of FAS^{3}T is solved by the classical all integer primal simplex method whose principle consists in moving from a feasible integer point to an other one until a feasible optimal solution is reached, by using the famous Gomory 's cut [Nemhauser and Wolsey 88].

4.5.2 FAS^{3}T behaviour with an example

The following concrete example (automatical vectorization of a Fortran loop, [Thomasset 87]) illustrates the behaviour of algorithm FAS^{3}T:

```
      Do 1 i=1,h
            il=n
            ip1=0
2           ip2=ip1
            ip1=ip1+il
            il=il/2
            i=ip1
            do 3 k=ip2+2,ip1,2
                  i=i+1
3                 x(i)=x(k)-v(k)*x(k-1)-v(k+1)*x(k+1)
            if( il . gt . 1)go to 2
1     continue
```

We consider the internal loop which is cut into pieces of length 256 (length of the vectorial registers of the actual computer). This normalization phase leads to the following code

```
      do  2  k5=1,1+(ip1-(2+ip2))/2,256
             k2=min(256,1+(ip1-(2+ip2))/2-k5)
             k3=-1+(ip2+k5)
             k4=-2+(2*k5+ip2)
             do 10 k1=1,k2,1
                   x(k1+k3)=x(2*k1+k4)-v(2*k1+k4)*x(-1+k4+2*k1)
     1             -v(1+k4+2*k1)*x(1+k4+2*k1)
10           continue
2     continue
```

By denoting

S_0= { k2 ≤ 256; k2 ≤ 1+(ip1-(2+ip2))/2-k5; k5 ≤ 1; k5 ≤ 1+(ip1-(2+ip2))/2-k5
k3 = -1+ip2+k5; k4 = -2+2k5 +ip2; 1 ≤ k5 ≤ k2; 1 ≤ kp1 ≤ k2; k1 < kp1}

the study of the data dependence leads to the consideration of three constraint satisfaction problems associated with the following systems :

S_1= { constraints of S_0; k1+k3 = 2kp1+k4; all variables are integer}
S_2= { constraints of S_0; k1+k3 = -1+k4+2kp1; all variables are integer}
S_3= { constraints of S_0; k1+k3 = 1 +k4+2kp1; all variables are integer}

For each system S_1,S_2 and S3, two iterations of FAS^{3}T prove the emptiness of the associated domain; this allows to conclude the possibility of vectorization of the considered Fortran loop.

4.5.3 Results

The following table details for each instance of system S:

the size (m : number of constraints; n : number of variables)
the number of non zero elements (N)
the answer to the CSP problem (empty if the emptyness of F(S) is proved; non empty is a solution of S is found)
the computational times in 1/100 th of seconds
by using the exact algorithm FAS3T, and

by solving directly but approximately the system S by an implementation of the simplex method (in this case, the instances with non decidability are pointed out with a star)

the ratio of these computational times .

m	n	N	answer	CPU times		ratio
				FAS3T	SIMPLEX	
10	6	15	empty	8	34	4.2
5	5	9	non empty	6	12	2
12	8	26	non empty	18	44	2.4
12	8	26	non empty	10	42	4.2
13	7	20	non empty	8	56	7
6	3	10	empty	2	8 *	4
6	3	10	non empty	4	10 *	2.5
19	11	29	empty	16	88	5.5
23	19	52	non empty	34	210	6.2
20	11	31	empty	18	158 *	8.7

For these instances the algorithm FAS3T is four times faster than an implementation of the simplex method directly applied to the initial systems. In addition it should be noted that this approximate solving leads to six cases of non decidability. On the other hand, the efficiency of our exact method is explained by the very small number of iterations, although it uses for each generic problem an all integer primal simplex method.

5 Conclusion

The two main characteristics of the general method FAS^{3}T for the solving of the CSP are

(i) it **solves exactly** each instance of the CSP: an integer solution of each system is found or the emptiness of the associated domain is proved. Thus the decidability is now exact with FAS^{3}T.

(ii) in addition, although the CSP is a NP-complete problem, the numerical experiments detailed in section 4 show that for the instances considered, FAS^{3}T is four times faster than an implementation of the simplex method directly applied to the initial systems.
Other experiments are in progress to study the effective impact of using FAS^{3}T in the vectorizer VATIL.

Références

Anderson R. , Bledsoe W.W. (1970), "A linear format for resolution with merging and a new technique for establishing completeness", J. ACM, pp 525-534.

Banerjee U. (1976) "Data Dependence in Ordinary Programs", MS Thesis University of Illinois at Urbana Champaign DCS Report No UIUCDCS-R-76-837.

Bennaceur H. (1989) , "Le problème de satisfaction de contraintes Synthèse et méthode exacte de résolution ", Thèse de Doctorat d'université Paris-nord.

Bennaceur H. et Plateau G. (1989), " Sur le problème de satisfaction de contraintes" , Rapport LIPN 89-6, université Paris-Nord.

Bennaceur H. et Plateau G. (1989), "An exact algorithm FAS3T for the contraints satisfaction problem", Proceeding of the workshop, "constraints processing and their applications", Detroit (USA).

Bledsoe W.W. (1974), "The SUP-INF methods in Presburger arithmetic", Research Report , ATP-18, Math. Dept., U. of Texas , Austin,Texas

Bledsoe W.W. (1975) "A new method for proving certain Presburger formulas" , 4 th Int. joint conf. on Artificial intelligence, Georgia(U.R.R.S.),15-21.

Gomory R.E. (1960), "All integer programming algorithm", IBM Research Center, Yorktown, Research Report RC 189.

Karmarkar N. (1984) "A new polynomial-time algorithm for linear programming" , Proc. 16th Annual ACM Symposium on Theory of Computing.

Nemhauser G.L. and Wolsey L.A. (1988)," Integer and combinatorial optimisation", Willey-Interscience.

Plateau G. (1979), "Contribution à la résolution des programmes mathématiques en nombres entiers ", Thèse de Doctorat d'état, Université des Sciences et Techniques de Lille.

Shostack R. (1977)," On the SUP-INF method for proving Presburger formulas ", J. ACM 24 (4) , pp 529-543.

Shostack R. (1981)," Deciding linear inequalities by computing loops residues ", J. ACM 28 (4) , pp 769-779.

Thomasset F. (1987), "Utilisation du calcul de Prédicats arithmétiques en vectorisation automatique", Rapport INRIA.

Wallace D.R. (1988) " Dependance of Multi-Dimentional Array References", Alliant Computer Systems, Inc. Littleton, Mass. 01460.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399